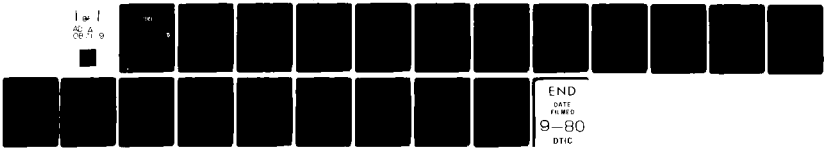
AD-A087 119 STANFORD UNIV CA COMPUTER SYSTEMS LAB F/G 15/3.1
DISTRIBUTED DATA PROCESSING FOR BMD.(U)
APR 80 M J FLYNN DAS660-77-C-0073
UNCLASSIFIED NL



END
DATE FILMED
9-80
DTIC

COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

DISTRIBUTED DATA PROCESSING FOR BMD

FINAL REPORT ~~ON CONTRACT NO.~~
DASG60-77-C-0073

APRIL 1980

SUBMITTED TO:

BALLISTIC MISSILE DEFENSE ADVANCED TECHNOLOGY CENTER

BY:

MICHAEL J. FLYNN

"Approved for public release: Distribution Unlimited"

DISTRIBUTED DATA PROCESSING FOR BMD

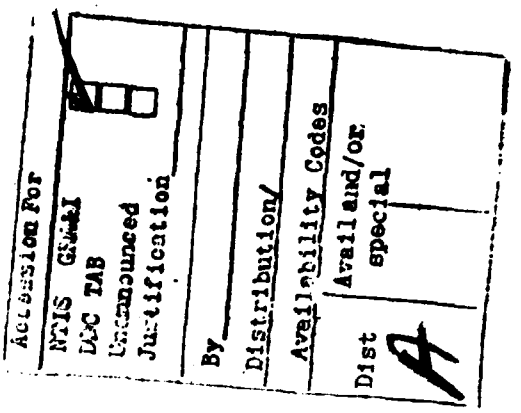
FINAL REPORT

Submitted to:
Director
Ballistic Missile Defense Advanced
Technology Center
ATC-P
P.O. Box 1500
Huntsville, AL 35807

From:
Michael J. Flynn
Professor, Electrical Engineering
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Contract No.: DASG60-77-C-0073

Period Covered: April 1, 1979 - March 30, 1980



FINAL REPORT

April 1, 1979 - March 30, 1980

DISTRIBUTED DATA PROCESSING FOR BMD

Michael J. Flynn

Principal Investigator

INTRODUCTION

Language oriented architecture provides an interesting approach to enhancing the performance of particular problem environments. Special language-oriented architectures called Directly Executed Languages (DELs) have been developed which are designed to provide very efficient program representations for particular languages and for particular classes of problems. In this report, section 1 is an introduction to these language oriented architectures and the second section is a review of some more recent work on Directly Executed Languages as they relate to BMD architecture development. Section 3 develops an introduction to a theory of language oriented operating systems to support the DEL concept.

All of these concepts were developed at the Stanford Emulation Laboratory, which provides a mechanism for measuring relative architectural performances. The Laboratory is described in the appendix. Section 4 of this report describes processor and architecture performance monitoring studies.

1. Language Oriented Architectures

Program forms or representations can be created which closely correspond to particular high-level languages. Such forms are called high-level language machines or in our terminology Directly Executed Languages. These are intermediate representations of original high-level language programs which are particularly suited for interpretation by a host machine under interpretive program control. The DEL representation is an intermediate form of the program, but even though it is not the high-level language itself, the DEL is directed at representing HLL objects rather than host objects. Thus, A * B → C appears as one DEL instruction, * and object names A, B, and C are coded with respect to their environment (scope each as a single object).

E.g. consider: A * B + C → D

LD	X=0	A
*	X=0	B
+	X=0	C
STO	X=0	D

(a) Traditional Single Accumulator Representation (X=0 means index unused)

*	A	B
+	C	D

(b) DEL code

Since each higher-level language has its own representation and interpreter, both operations and operands are interpreted as defined by the higher-level language. The instruction vocabulary is exactly the same as the HLL actions. The data types available (interpretative) again correspond to those defined in the HLL. Careful DEL design will avoid the introduction of any object not present in the HLL representation; i.e. no temporaries, etc. In the above, a

powerful format set associated with the OPS (operation) allows the result of the * to be a source of the + operation. However, the DEL is not the HLL -- values either are or can be readily bound to operand names. Coding of objects is not mnemonic; rather, it is very concise. Object container sizes can be restricted to $\log_2$ of the number of such objects in the HLL scope of definition-- usually a significant savings over host oriented identifiers. Notice in the above example the traditional machine includes an index field with each instruction identifier, even though in this particular example it is not used. The DEL identifier on the other hand should be regarded as a concise pointer to a more detailed data descriptor of the object to be used. A quasi-ideal representation for Fortran was developed called DELtran and further work continues on DELs for Pascal, Algol and Cobol. The performance of DELtran on the host Emmy is interesting when compared to System 370 performance on the same system. The table below is constructed using a version of the well known Whetstone benchmark for Fortran machine evaluation. The Emmy host system is a very small system-- the processor consists of one board with 305 circuits modules and a 4096 32-bit word interpretive storage. It is clear that the DELtran performance is significantly superior to the 370 in every measure.

DELtran vs System 370 Comparison for the Whetstone Benchmark

Whetstone Source -- 80 statements (static)
-- 15,233 statements (dynamic)
-- 8,624 bits (excluding comments)

	System 370 FORTRAN-IV opt 2	DELtran	ratio 370/Deltran
Program Size (static)	12,944 bits	2,428 bits	5.3:1
Instruction Executed	101,016 i.u.	21,843 i.u.	4.6:1
Instruction/Statement	6.6	1.4	4.6:1
Memory References	220,561 ref.	46,939 ref.	4.7:1
EMMY Execution Time (370 emulation approximates 360 Model 50)	0.70 sec.	0.14 sec.	5:1
Interpreter Size (excludes I/O)	2,100 words	800 words	2.6:1

Experience with the DELs for Fortran show static size improvement of 5 to 1 in required representation space and about 4:1 improvement over the number of instructions to be interpreted when compared with traditional host oriented instruction sets.

A simple DEL implementation model is shown in Figure 1. The DEL program representation lies in image store (main memory) while the interpreter and the interpreter parameters lie in a special high speed interpretive storage (corresponding to a read/write microprogram store). The host is unbiased with respect to any image instruction set--i.e. it is a special purpose machine designed for high speed interpretation.

The resulting arrangement is called soft architecture. The representation to be interpreted depends completely upon the interpreter. Non-dedicated host machines (not dedicated to any particular image) incur some overhead due to their inherent flexibility. However, this is more than compensated for by the smaller number of objects required to be interpreted in a DEL program representation. Moreover, specially dedicated host machines could be defined which would more closely correspond to a particular HLL and its environment, avoiding in this case even this interpretation overhead.

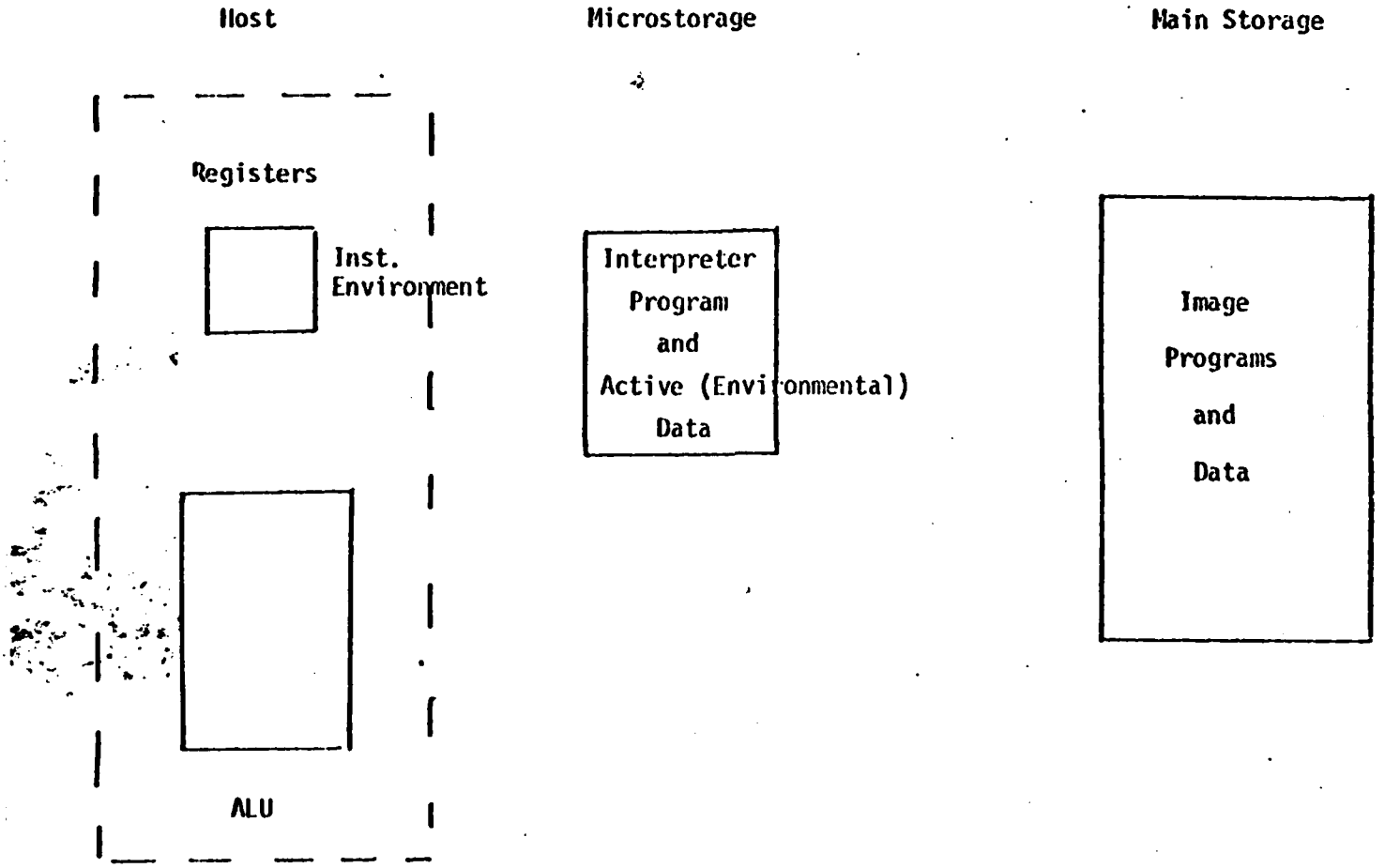


Figure 1. DEL Storage Assignments

2. Directly Executed Languages and "Ideal" BMD Architecture Development

In order to apply DEL concepts to BMD applications further compiler development was accomplished. The DEL compiler was restricted permitting analysis of tpred, one of the Site Defense subroutines. We soon discovered tpred is a very unusual piece of code. Most of the variables that occur in the code itself are arrays, and the subscripts of these arrays as used are constant throughout the routine. This allows a great deal of optimization of address calculation, something that the compiler does not yet do. Therefore, we analyzed tpred by hand, with the results shown below:

Static size in bits

IBM 360		DEC 20		DEL	
non-optimized:	17688	non-optimized:	19440	non-optimized:	5128
optimized:	12352	optimized:	19188		

Static number of instructions

non-optimized:	721	non-optimized:	540	non-optimized:	248
optimized:	450	optimized:	533		

The problem of translating code involves flow of control statements. This creates a problem unique to DEL compilation involving dependent instructions. As the compiler generates each field of the DEL code stream, it attempts to pack it into a word of memory. When a field doesn't fit, the compiler pads the word with zeros and puts the field at the beginning of the next word. If the contents of a field are not known, the compiler can reserve space for the field provided it knows the proper width needed. Compilation can proceed and it can fill in the unknown field later. If the width is not known then filling in the field will require unpacking all successive words and repacking them by again testing for fit and padding the ends of words. Unpacking will require parsing many of the fields to determine the format of each instruction, that is, how many fields each

instruction consists of and how wide each field is.

Consider one case, however, in which the width of a field is not known at the time it is generated but even so, filling it in later does not require unpacking and repacking all successive words. This is the case when the possible widths of the unknown fields are equal to a constant plus some multiple of full word width. For example, suppose the field is either eight bits wide or eight bits and one word wide. On encountering this field, the compiler can generate an eight bit wide dummy field. Later, when it knows the field's width and contents, it can fill it in and, if needed, insert the extra full word, moving succeeding words down one address without any unpacking or repacking.

The branch instructions in our DEL are span dependent. If the goal of the branch relative to the program counter (the span of the instruction) can be expressed in eight bits, then the compiler generates a short branch instruction; otherwise it generates a long one. By specifying long branch instructions to be eight bits plus one word wide, we allow the compiler to avoid some extra work at the price of a slightly long branch instruction.

To verify parts of the compiler, especially arrays and flow-of-control, two kinds of instructions were added to the interpreter and the Quicksort program was run. A pseudo-random number generator and an outer loop to execute the program 1000 times was added and the resulting program was run on various machines.

Relative Measure	DELtran	DEC-20	P-Machine
Static no. of instructions	1	2.8	4.7
No. of instructions executed	1	1-2	4.6
Time to execute on Emmy	1	-	3.3

(The P-Machine is a Pascal P-Code interpreter for the Emmy. See TN164, September 1979.)

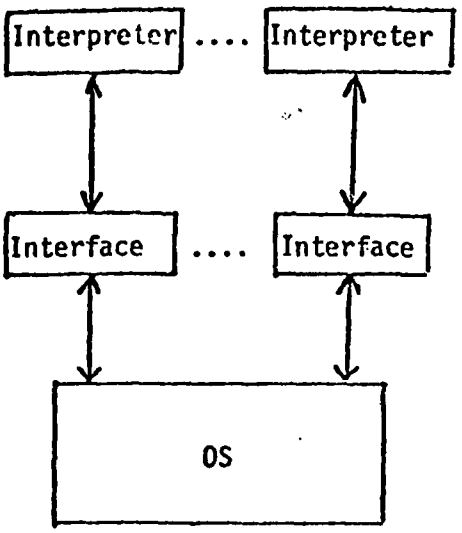
3. Language-Oriented Operating Systems

Traditional language interpreters supported by conventional operating systems usually require relatively elaborate interfaces to match the needs of the interpreter with the services supplied by the operating system. In a multi-language environment each language interpreter requires a different interface. The operating system in this case is not directly integrated into the language interpreter (Figure 2a).

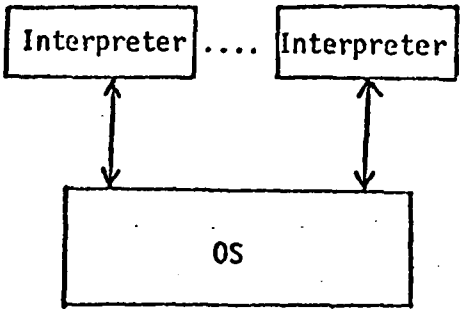
In a language-oriented operating system, however, the choice of system facilities is driven by the given language. In a multi-language environment, the operating system must be flexible enough to provide support for many different language features. For instance, a Fortran-like language would probably be more demanding--e.g. support of concurrent processes. In any case, the facilities required by each language must be taken into consideration in designing the operating system.

We believe that there are a set of concepts, structures and mechanisms that are common to higher-level languages, and that these elements can be embodied in an operating system which directly supports the (concurrent) operation of different language interpreters (Figure 2b).

A primary concept in developing such a system is that of an atomic function (or action)--a non-interruptable process providing support for language primitives. For example, the language primitive WRITE in Fortran might have as one of its associated atomic functions the initiation of a channel command. Such language primitives can be realized with sequences of atomic functions, interrupts being serviced at atomic boundaries.



(a) Matching Needs to OS



(b) OS Support Influenced by Interpreter Needs

Figure 2

In designing and creating atomic functions it is desirable to achieve as sizable an execution time as is possible in order to minimize the relative significance of entry and exit overhead during interpretation. However, peripheral device characteristics require that interrupts must be serviced by a specified time after their occurrence or information will be lost as well as, perhaps, opportunities for faster operation. Thus atomic functions should be selected so that their execution times fit within a fixed time envelope.

A system structured by this approach has several advantages including (1) a more transparent system design (e.g. providing direct support for language primitives), (2) the opportunity for efficient operation by minimizing the number of objects interpreted, and (3) the natural protection mechanisms inherent in an interpretive approach.

An opportunity exists with a DEL structure based upon atomic functions for a unification of the concepts in higher-level language design and operating system construction. Clearly there exists a DEL for any given higher-level language, and this DEL sits at the interface between the user and the hardware. There can also be a DEL for the operating system which sits at the interface of the language interpreter and the operating system support. This DEL would consist of the atomic functions.

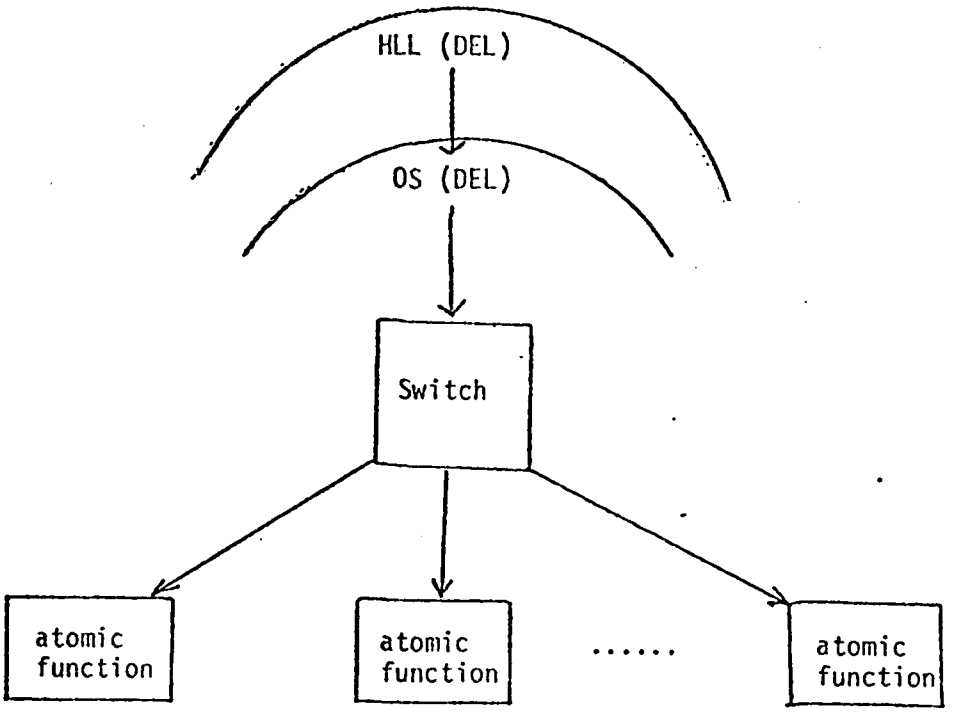


Figure 3. Logical Realization of an OS DEL

At this lower level, there could be a switching mechanism which sequences through the atomic functions associated with a given request from a language primitive. In a sense, the switch is a form of logical control unit and the atomic functions are a form of logical functional units. (see Figure 3). This switch may be realized in many ways. It could take the same form as the control for the higher-level language DEL (e.g. dynamic contour) or could take the form of the Supervisor detailed in.

Protection

The protection of users and programs is natural in this type of system. As an example, let us assume that the interpretive system is realized in two levels:

1) the language interpreter level (level 2)
2) the OS support level (level 1)

Protection of the address space and procedures of the language is easily embedded in the language interpreter. In fact, a great deal of protection is afforded in most higher-level languages by the unavailability of the use of the idea of a storage address. A microprogrammable host machine makes it possible for the implementation of protection mechanisms to be efficient.

Protection is also possible in the interface between level 2 and level 1. System support functions can be addressed by capabilities. A capability is a special kind of address for a virtual object that can only be created by the system. In order to use the object the capability provides accessing information. Of course the advantage of a capability scheme is that access to a system facility can be checked without actually handling the system object----i.e. checking can be accomplished entirely at level 2.

4. Processor Performance Monitoring

One of the primary missions of the Emulation Laboratory is to provide a base for measuring the performance of processor architectures. Recently we have completed one phase of this by monitoring and evaluating a target processor in a sophisticated instruction environment. Our approach is to instrument a functioning emulation system with microcode based event counters. During real time operation of the emulator these counters record the freqnency of various internal events of interest.

Monitoring of the PDP-11 [TN 156] is our first major endeavor in this area. The PDP-11 emulation is capable of fully supporting mini-UNIX, a widely used operating system. Event counters collect data about processor operation in the following areas.

(1) Opcode Usage

(2) Operand Usage

(3) Qualifier usage (i.e. branch offset)

(4) Dynamic branch outcome

A typical measured experiment involves tracking a program from initialization to completion and usually involves the execution of several million instructions. Provisions have been made to control monitor enabling from the image machine. This allows measurement of either user code, system code or both. The ability to measure system code is a unique feature of our laboratory environment.

Data from the laboratory is currently analyzed on a remote processor system. This analysis examines the measured instruction stream from two standpoints, syntax and semantics. The syntax of an instruction stream is the way in which instruction words are assigned to various functions such as operation, operand and data specification. Instruction stream semantics is concerned with

the measuring of bit assignments and their influence on processor resource usage.

Syntactically, we can give an exact picture of instruction word usage in the PDP-11. Each PDP-11 instruction consists of 21.66 bits with the following usage:

	COMPONENT	BITS	
BASE	OPCODE	6.61	
	OPERAND	7.51	
	QUALIFIER	1.89	(branch offset)
		16.00	
EXTENSION	INDEX	1.49	
	INDEX DEFERRED	0.12	
	IMMEDIATE	2.06	
	ABSOLUTE	0.25	
	RELATIVE	1.68	
	RELATIVE DEF	0.06	
		5.66	

AVERAGE INSTRUCTION LENGTH = 21.66 BITS

PDP-11 Instruction Syntax

From a semantic point of view we can use information gathered by the monitor to describe processor memory and resource usage. Below we show accesses on a per instruction basis which the PDP-11 processor makes to its two primary memory resources: register and memory

ACCESS	REGISTER	MEMORY
INSTRUCTION		1.000
DISPLACEMENT		0.209
DATA READ	0.329	0.453
DATA WRITE	0.292	0.220
ADDRESS	0.706	0.040
MISC READ	0.075	0.033
MISC WRITE	0.075	0.050
ALL READS	1.110	1.735
ALL WRITES	0.366	0.270
TOTAL	1.476	2.005

PDP-11 Storage Resource Usage

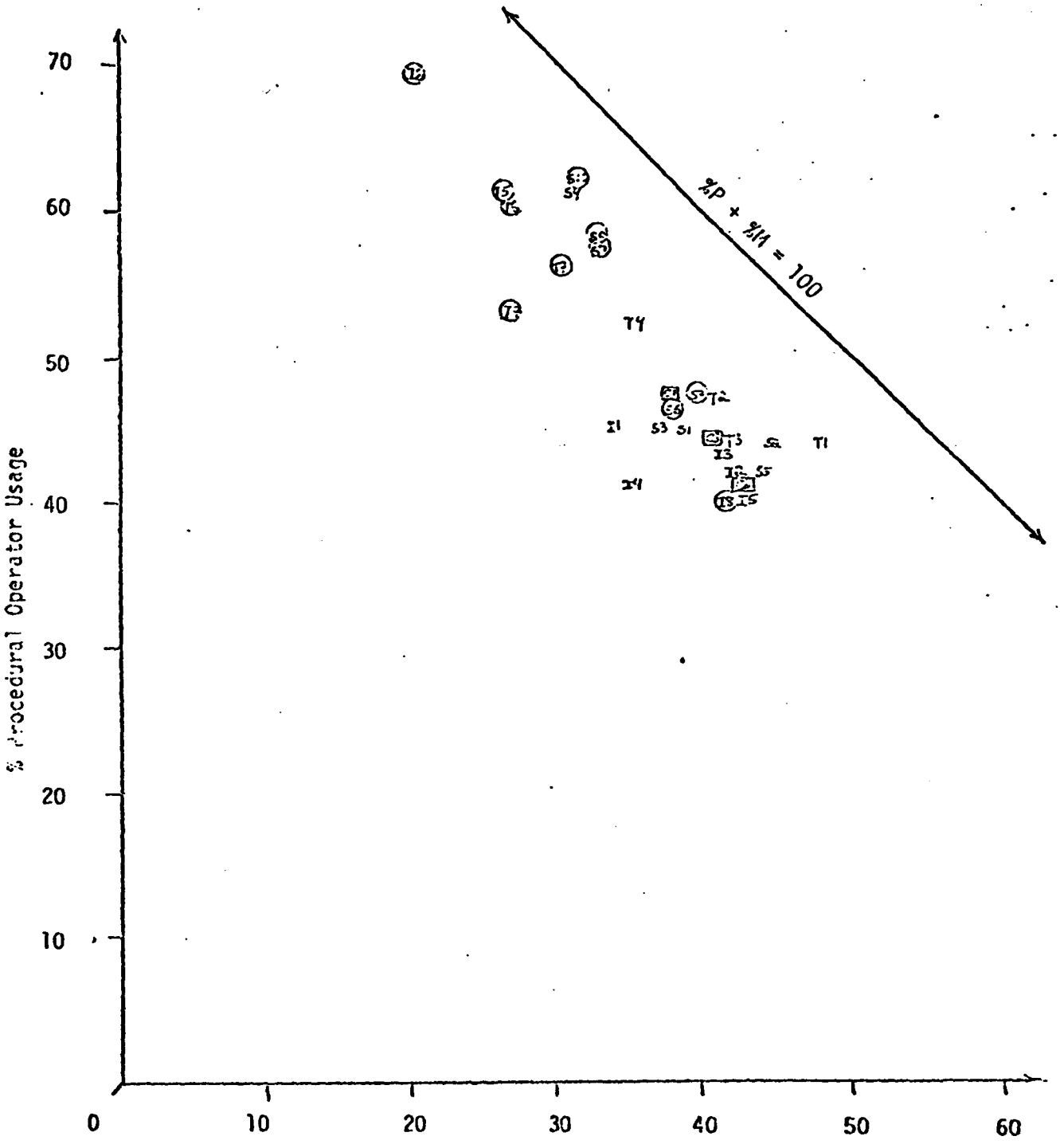
One measure of processor effectiveness which we have employed previously in to classify instructions into three groups:

F - Functional -- Arithmetic, logical, shift

M - Memory -- Move

P - Procedural -- Test, compare, branch

Using the monitored PDP-11 we have measured the fraction of processor opcodes in each category for a group of 28 programs. These measured programs span four areas: translation, information organizing, system oriented (interactive) and computation. The tested programs total 90 million instruction executions. Three source languages, "c" assembler and Fortran, are represented. Figure 4 shows the results from the tested programs. As can be seen, the operating region for the PDP-11 architecure is confined to a small area whose characteristics are small F operator usage and large M and P usage. We are currently investigating alternative processor architectures which would reduce M and P usage and thereby increase processor effectiveness [TR170, 171].



□ - Fortran Source
○ - Assembly Source
⌓ - "C" Source

T# - Translation
I# - Information Organizing
S# - System
C# - Computational

ISSUE	ENGR / DRAWN	TITLE	SHEET
		Fig. 4: Operator Usage for 28 Tests	NO. OF SHEETS PER SET

APPENDIX

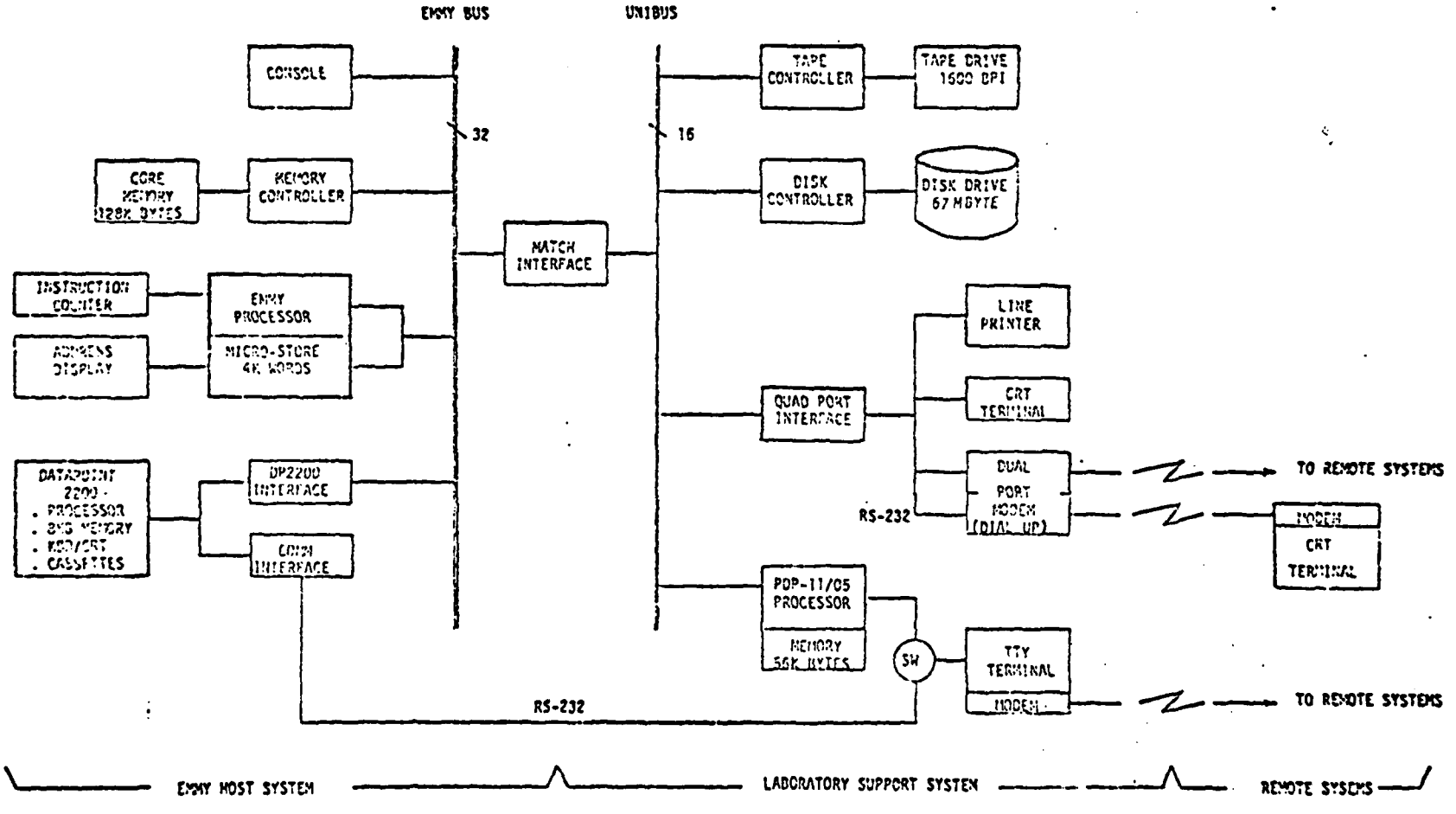
The Emulation Laboratory

Figure 5 (attached) shows the current configuration of the laboratory. A PDP-11/05 processor acts, under the UNIX operating system, as the I/O processor and emulator for image machines under emulation. All I/O and operating system commands are mapped through UNIX and handled by this device. Image tapes are mapped onto our disk while the lab's tape drive is used largely for archival data storage. Our specially designed Emmy processor emulates the machine instructions (except the I/O instructions) of the image processor. A Datapoint is available to act as a console for the machine under emulation (the image machine) and communication modems are available for external users of the system. A number of image emulators have been written for Emmy. These include:

1. PDP-11
2. IBM 360
3. CDC 6600
4. Data General Nova
5. Intel 8080 (MDS)
6. RC 4000
7. Hewlett Parkard 2100

In order to support the writing of image machine emulators several software toos were developed. They included an assember called EMMY/XL and a higher level macro assembly language called EMMY/PL. This software plus a basic operating system for Emmy and the recent addition of a virtual input/output system has significantly simplified the problem of emulator production.

One of the principle purposes of the Emulation Laboratory was to perform cross architectural comparisons, that is, to compare image machines and to determine architectural aspects which lead to more efficient program representations. This effort includes the development of the emulators, their verification and (most significantly) the handling of the I/O and operating systems functions to accomodate a statistically significant corpus of programs for evaluation.



EMULATION LABORATROY

Figure 5

I/O Device Emulation

Our initial use of the emulation laboratory was to support the interpretation of various processor architecture in a "code only" environment. In order to investigate more realistic environments we have complemented the code emulator, based in Emmy, with a general purpose I/O device emulator, based on the PDP-11 laboratory support (see Figure 5). The device emulation system is called Vaccess (for Virtual Access) [TN. 114 and "I/O Device Emulation in the Stanford Emulation Laboratory" by J. Huck and C. Neuhauser, published in the Micro 12 Proceedings]. There are two problem areas that must be addressed by the device emulator design. First, the device emulator must support a wide range of device structures and characteristics. Systems may require such varied devices as disk, tape, line printer and cassettes. Second, provisions must be made for users to manipulate the physical environment of the device. An example of this would be mounting tapes or changing disk packs.

The Vaccess program runs as a processor under the UNIX operating system on the laboratory support processor. Operationally, Vaccess acts as an I/O channel handling data transfer requests from the code emulator based on the Emmy processor. Requests from the code emulator transfer a block of data from Emmy storage to a designated virtual device. Because Vaccess can make use of all the facilities of UNIX these data transfers may be directed arbitrarily to files, disk storage or actual physical devices. For example, output from an emulated line printer may be sent to a disk file in UNIX and later examined from a terminal.

We have found that three basic device structures are capable of handling a majority of the peripheral unit structures encountered in system emulation:

(1) Serial - character oriented streams to emulate terminals, line printer, etc.

(2) Linear - Randomly accessible, one dimensional, arrays used to emulate disks, drums and diskettes.

(3) Variable - A collection of variable length records for emulating tape units or blocked storage.

Device structures are generally specified with particular device features (e.g. block size) imposed by the code emulator via request parameters. Vaccess provides facilities for controling virtual to real device mapping, interactive request debugging and status indication.

At present five emulators have been constructed which utilize the Vaccess system. Device assignment and characteristics are shown below.

	--- Serial ---				Linear	Variable	
	TTY	PPT	LP	CR	DISK	TAPE	Supports
S/370	-	-	Y	Y	-	Y	PL360 OS, Fortran, Pascal
PDP-11	Y	Y	Y	-	10 Mb	-	Mini-UNIX
NOVA 3	Y	Y	Y	-	10 Mb	-	NOVA RDOS
MDS 8080	Y	Y	Y	-	-	-	Simple programs
P-machine	Y	-	-	-	-	Y	Pascal programs

Table 1: Complete Emulation Systems

Data transfer rates to emulated devices range from 1K bytes per second for single character transmissions to 60K bytes per second for blocked data to bulk disk. These transfer rates have been found adequate to support emulation of interactive systems such as UNIX in a usable manner.